6.826 Principles of Computer Systems	Fall Semester, 1993

Lecture 12: October 25, 1993

Lecturer: Bill Weihl — *Scribe: Yan Z. Ding & Greg Kimberly*[1]

1 Note About Addresses

Mail concerning questions on the problem sets should be sent to both TA's or to

`6826-staff@jukebox.lcs.mit.edu`

Messages intended for everyone taking the class should be sent to

`6826-forum@jukebox.lcs.mit.edu`

2 Lecture Overview

Today's lecture begins a two-part discussion on transactions. The topics covered are:

- Sequential Transactions
- Performance Considerations
- Sequential Transactions with Caching
- Recovery

3 Sequential Transactions

Transactions provide a programmable interface for making sequences of actions atomic. Sequential transactions, which are covered in this lecture, enable the writing of big atomic actions with respect to failure. Concurrent transactions extend the atomicity to concurrent access.

4 Performance

The `SequentialTransaction` spec and the `LogRecovery` implementation were presented in Handout 10 and repeated in Handout 31. There are two important metrics for transactions system performance:

- latency for each transaction

[1] These notes are based on notes prepared by Susan Yeh in Fall '92

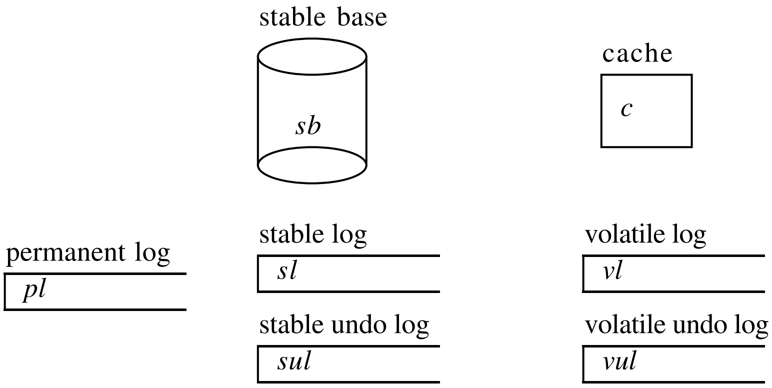

Figure 1: Elements in Sequential Transactions with Caching

- time to execute commit

Both of the above implementations perform poorly under these measures. We add mechanisms for improving performance:

- Cache information on the volatile state (`vs`). Since the entire volatile state is probably too large, we construct `vs` from the stable state plus what is in the cache. By caching frequently used parts of `vs`, updates and reads can be faster.

- Decouple commit from actually applying the updates written on the stable log to the stable base. This reduces the latency of the commit. A fringe benefit is that we can aggregate a series of updates to a single disk block and pay only one disk write to record them.

5 Sequential Transactions with Caching

The basic ideas for implementing sequential transactions with caching are:

- Since the cache will be of some finite size, we can't allow the cached updates to grow without bound. This means that we'll have to flush the cached updates when it is determined that the cache is full. Since the updates are flushed independent of whether the containing transaction has committed, the Stable Base may include updates not yet committed.

- We want to defer updates for as long as possible, as we can aggregate many updates of a single disk block and then just apply the most current one to the Stable Base. As mentioned above, however, by flushing the cache as it fills, the Stable Base can contain updates not yet committed. Recall that the abstract Stable State must not contain uncommitted updates. To restore the Stable State from the Stable Base (which contains committed and uncommitted updates), we'll need to calculate an "undo" action for each action, so that any uncommitted actions can be undone:

  `ss = sb +` All updates − uncommitted updates (undo)

- We want to truncate the log of entries whose updates have been applied to the stable base in order to reuse log space. We'd like to truncate entries whether or not the transaction responsible for them has committed (as we've decoupled commit from the application of updates)—so long as they've been applied to the `sb`, we can remove them from the log.

- To enable recovery of the base in case the disk gets corrupted, we'll need to save discarded entries (those truncated in the above process) to a Permanent Log (`pl`). The `pl` is often kept on magnetic tape.

5.1 Failure Types

There are four levels of failure within this transaction subsystem, listed in order of increasing severity:

- **Transaction Abort**: the client aborts a transaction in progress. It is not really a failure, since no work is lost except by request.

- **Crash**: this results in the loss of the volatile state and aborts all active uncommitted transactions. Committed transactions are recoverable from the Stable Log and Undo Log.

- **Media Failure**: this results in the loss of the Stable Base, as well as the Stable Log and Stable Undo Log. The Stable Base is recoverable from the Permanent Log.

- **Catastrophe**: the Permanent Log and the Stable Base are lost.

In practice, further steps are taken to handle failures, such as periodically taking snapshots of the Stable Base and saving multiple copies to magnetic tape.

5.2 Implementation: LogAndCache

Handout 31 presents an implementation for sequential transactions with logging and caching. The rest of the lecture highlighted the important features of the parts of the implementation. The implementation has the following features:

- An action is turned into a sequence of updates that can be applied atomically. Updates are applied to single disk blocks (which can be written atomically).

- Undo Actions: for every action there exists a corresponding "undo" action which will reverse the effect upon the base of that action.

- The redo and the undo entries of an action are written to the logs in one atomic step.

- A cached update, or W, represents the new contents of a disk block.

The implementation is optimized for both crash recovery (due to logging) and for speed of access (due to caching). The system now looks like Figure 1 which contains the following parts:

- Stable Base
- Cache
- Stable Log (`sl`) and Volatile Log (`vl`)
- Stable Undo Log (`sul`) and Volatile Undo Log (`vul`)
- Permanent Log—which stores discarded parts of the Stable Log and can be used for emergency reconstruction of the stable base.

An action is converted to a sequence of updates (U), which are logged to the `vl`. The updates are also applied directly and immediately to the cache (`c`) before commit, so that the current state is immediately accessible. For each action, an Undo Action is computed and written to the Volatile Undo Log (`vul`). Note that the undo log contains the action, and not the sequence of updates into which the action is converted—this is done for efficiency, as we normally don't expect to have to undo actions, and so we can defer converting them to updates if and when we apply them. (For concurrent transactions you must undo the action as updates may not commute with changes applied by other transactions.) Committing a transaction first moves its updates (U) from the `vl` to the `sl` and its undo actions from the `vul` to the `sul`. It then logs a commit record in the `sl` and empties the `sul` (since the action is committed, we no longer need to be able to undo it).

As noted in Handout 31, the U's and W's have the following properties:

- Adding an action's associated updates (U's) to the log is an atomic operation.
- Applying a cached update (W) to the `sb` is an atomic operation.
- Converting a U to a W is cheap, and applying a W is cheap.
- U's are idempotent (the hiccup property as discussed in Handout 10).
- W's commute

We can write an abstraction function to *SequentialTransaction* from Handout 10:[2]

```
ss = sb + sl - sul
vs = sb + sl + vl
```

5.3 Implementation Routines

The important aspects of many of the implementation routines were examined:

[2] As noted in Problem Set #7, this abstraction function is incorrect. The correct one is provided in the solution to that problem set.

- **DoOrUndoA**: We first construct the volatile state of the system (`vs`) from the `sb` and the cache by applying all cached updates (W's) to the stable base. We then convert the action to its sequence of updates based upon the volatile state. If the action is not an undo action, then we compute an undo action. If we were performing an undo action, then we need to log the fact that this undo action has actually been performed. The undo action is broken into updates, which are logged in the redo log just as a normal action's updates would be. A cancel record is logged in the undo log; this provides idempotence in the undo log without requiring the undo actions to be idempotent.

  The action's updates are logged to the `vl`, while an undo action is logged to the `vul` and the updates are converted to cache updates (W's) and applied to the cache. As specified, the entire procedure is done atomically.

- **Commit**: Commit first forces all updates from the `vl` to the `sl` and all undo actions from the `vul` to the `sul`. It then logs a "commit" marker to the `sl` and empties the `sul`, doing these two operations atomically.

- **Abort**: Abort undoes all uncommitted updates by applying the undo actions in the undo logs. In the implementation given in the handout, it forces all updates in the `vl` and `vul` to the `sl` and `sul`, respectively. This is not necessary in general because in the event of a crash, the transaction will be aborted anyway.

- **Crash**: The idea of Crash is that the volatile state has been lost. This includes the `vl`, the `vul` and the Cache (`c`). Recovery then consists of replaying the `sl` to reconstruct the cache as it existed before the crash, and then applying any undo actions contained in the `sul` to undo all uncommitted updates.

- **Undo:** Undo concatenates the `sul` with the `vul` and then applies the undo actions contained in this concatenated log *in reverse order*. The undo actions must be applied in order, and since each was computed based upon the volatile base that existed after the application of the previous undo action's associated action, this requires a reverse chronological order. Undo skips over previously applied undo actions until one is encountered which has not yet been applied. It is then applied by invoking `DoOrUndoA()`, which in addition to converting the undo action to a sequence of updates, also marks the undo action as applied in the `vul`. In this way undo actions are only applied once, assuring idempotence of the undo logs.

- **Redo**: Redo rebuilds the cache as it was before the crash, thus containing volatile uncommitted updates. This is faster than rebuilding and applying all updates to the `sb`, and uncommitted updates can be gotten rid of via `Undo()`.

- **Flush**: flushes an update from the cache to the stable base, to make room in the cache. The *write-ahead log* policy requires that the corresponding update in the log (and the corresponding undo action) must be forced to the disk *before* the update from the cache is allowed to change the stable base. This ensures that in the event of a crash, the update in the stable base will be undone.

Before applying a cached update to the `sb`, Flush makes sure that the corresponding action is in the `sl`.

- **Truncate**: Truncate takes a contiguous portion of the `sl` that has already been applied to the `sb` and removes it from the `sl`. If protection from media failure is desired, the truncated portion of the `sl` is added to the permanent log, which can later be used to re-create the stable base.